# Dynamic Symbolic Execution of Distributed Concurrent Objects *

Andreas Griesmayer[1], Bernhard Aichernig[1,2],
Einar Broch Johnsen[3], and Rudolf Schlatte[1,2]

[1] International Institute for Software Technology, United Nations University
(UNU-IIST), Macao S.A.R., China
{agriesma,bka,rschlatte}@iist.unu.edu

[2] Institute for Software Technology, Graz University of Technology, Austria
{aichernig,rschlatte}@ist.tugraz.at

[3] Department of Informatics, University of Oslo, Norway
einarj@ifi.uio.no

**Abstract.** This paper extends dynamic symbolic execution to distributed and concurrent systems. Dynamic symbolic execution is used to systematically identify equivalence classes of input values and has been shown to scale well to large systems. Although mainly applied to sequential programs, this scalability makes it interesting to consider the technique in the distributed and concurrent setting as well. In order to extend the technique to concurrent systems, it is necessary to obtain sufficient control over the scheduling of concurrent activities to avoid race conditions. Creol, a modeling language for distributed concurrent objects, solves this problem by abstracting from a particular scheduling policy but explicitly defining scheduling points. This provides sufficient control to apply the technique of dynamic symbolic of interleaved processes. The technique has been formalized in rewriting logic and executes in Maude.

## 1 Introduction

Distributed and concurrent systems, e.g. web services, are becoming increasingly important for long-running infrastructure and applications. They typically consist of loosely coupled components which communicate asynchronously, potentially running on different hardware systems. For critical distributed systems, the use of formal methods, both for design and verification, remains a challenge. In the general case, the complexity of such systems makes full verification seem impossible, even for medium sized examples.

The challenge is to find a verification technique that scales to the combinatorial explosion in the number of possible runs in such models. A promising

* This research was carried out as part of the EU FP6 project *Credo*: Modeling and analysis of evolutionary structures for distributed services (IST-33826).

technique that seems to scale well to large systems is *dynamic symbolic execution* [1, 4, 9]. The idea is to calculate a symbolic execution in parallel with the concrete test run of a given formal model. The result is a set of conditions over symbolic input values representing the path of the last run.

The problem is that dynamic symbolic execution is of limited use with the concurrency models of today's programming languages. The reason is that dynamic symbolic execution does not work in settings where the execution of expressions is not atomic. Hence, its main application so far has been limited to single-threaded programs and to client-server applications with simple serialized communication flows. In this work we overcome this limitation by choosing a modeling language that provides the appropriate level of concurrency control: Creol [6], an executable object oriented modeling language whose execution model was designed to assist in the development of distributed systems.

We have implemented the dynamic symbolic execution technique in Maude [2], which is the execution platform of Creol, allowing us to perform the symbolic run dynamically while the concrete run is executed. The tool, and an application to testing, is covered in more detail in [5].

### 1.1 Related Work

Symbolic execution is a widely used program analysis technique that represents the values of variables as symbolic expressions instead of concrete data. An execution of a program is performed by manipulating those expressions instead of computing concrete values.

Application of symbolic execution to verification was already proposed in 1976 by King [7], who shows symbolic execution for a simple sequential language and presents an interactive tool EFFIGY to traverse the execution tree. However, there are limits to the feasibility of this technique, due to the sheer number of possible execution paths induced by non-determinism. To make the process feasible for large systems one can either reduce the amount of information that is tracked, or the number of paths to search. An example for the first kind are static analysis tools like ARCHER from Engler et al. [10], which concentrate on certain properties of interest for the analysis. In this paper, we reduce the number of paths that are searched at a time by *dynamic symbolic execution*. In [1], Boyer et al. show the interactive tool SELECT that computes input values for a run selected by the user. One of the first automated tools was DART from Godefroid et al. [4], which automatically extracts a program's interface and generates a test driver to perform random testing. Several extensions to these approaches exist, among the most notable the PEX tool from Tillmann et al. [9] for creating *parameterized unit tests* for single-threaded .NET programs. We extend dynamic symbolic execution to Creol's concurrency model, including the treatment of local scheduling points in the distributed objects.

## 2 The Modeling Language Creol

Creol is a high-level executable modeling language targeting distributed systems in which concurrent objects communicate asynchronously [6]. The language decouples communication from synchronization. Furthermore, it allows local scheduling to be left underspecified but controlled through explicitly declared process release points. The language has a formal semantics defined in rewriting logic [8] and executes on the Maude platform [2]. In the remainder of this section, we present Creol and point out its essential features for DSE.

A concurrent object in Creol executes a number of processes that have access to its local state. Each process corresponds to the activation of one of the object's methods; a special method `run` is automatically activated at object creation time, if present, and captures the object's active behavior. Objects execute concurrently: each object has a processor dedicated to executing the processes of that object, so processes in different objects execute in parallel. In contrast to, e.g., Java, each Creol object strictly encapsulates its state; i.e., external manipulation of the object state happens via calls to the object's methods only. Only one process can be active in an object at a time; the other processes in the object are *suspended*. A process can be *released* using Creol's **`await`** statement, in which case another proces may be activated.

Communication in Creol is based on method calls. These are a priori asynchronous; method replies are assigned to labels (also called *future variables*, see [3]). There is no synchronization associated with *calling* a method. *Reading a reply* from a label, however, is a blocking operation and allows the calling object to synchronize with the callee. A method call that is directly followed by a read operation models a synchronous call. Thus, the calling process may decide at runtime whether to call a method synchronously or asynchronously.

The language syntax of the subset of Creol used in this paper is presented in a Java-like style. We omit some features of Creol, including interfaces, inheritance, non-deterministic choice and many built-in data types and their operations. For a full overview of Creol, see for example [6].

### 2.1 Representation of a Run

A run of a Creol system captures the parallel execution of processes in different concurrent objects. Such a run may be perceived as a sequence of execution steps where each step contains a set of local transitions on a subset of the system's objects. However, only one process may be active at a time in each object and different objects operate on disjoint data. Therefore, the transitions in each execution step may be performed in a truly concurrent manner or in any sequential order, as long as all transitions in one step are completed before the next execution step commences. For the purposes of dynamic symbolic execution the run is represented as a sequence of statements which manipulate the state variables, together with the conditions which determine the control flow. Due to space restrictions, we concentrate on statements for the concurrency model, namely asymchronous method calls and await statements. The representation of the other statements is straight forward and can be studied in more detail in [5].

An asynchronous method call in the run is reflected in four execution steps (the label value $l$ uniquely identifies the steps that belong to the same method call): $o_1 \overset{l}{\rightharpoonup} o_2.m(\overline{e})$ represents the *call* of method $m$ in object $o_2$ from object $o_1$ with arguments $\overline{e}$; $o_1 \overset{l}{\rightharpoondown} o_2.m(\overline{v})$ represents when the called objects starts execution, where $\overline{v}$ are the local names of the parameters for $m$; $o_1 \overset{l}{\leftharpoonup} o_2.m(\overline{e})$ represents the emission of the return values from the method execution; and $o_1 \overset{l}{\leftharpoondown} o_2.m(\overline{v})$ represents the corresponding reception of the values. These four events fully describe method calling in Creol. In this execution model the events reflecting a specific method call always appear in the same order, but they can be interleaved with other statements.

Conditional statements in Creol are side effect free and therefore only represented in form of the statements of the branch that was actually executed. For the sake of computing the input values, however, the condition of the taken branch is recorded as $\langle g \rangle$. Remark that statements **`await`** $g$ requires careful treatment: if it evaluates to *false*, no code is executed. To reflect the information that the interpreter failed to execute a process because the condition $g$ of the **`await`** statement evaluated to *false*, the negated condition $\langle \neg g \rangle$ is recorded.

## 3 Dynamic Symbolic Execution of Distributed Objects

Conventional symbolic execution uses symbols to represent arbitrary values during execution. When encountering a conditional branch statement, the run is forked. This results in a tree covering all paths in the program. In contrast, dynamic symbolic execution calculates the symbolic execution *in parallel* with a concrete run that is actually taken, avoiding the usual problem of eliminating infeasible paths. Decisions on branch statements are recorded, resulting in a set of conditions over the symbolic values that have to evaluate to *true* for the path to be taken. We call the conjunction of these conditions the *path condition*; it represents an equivalence class of concrete input values that could have taken the same path. Note, in the case of non-determinism, there is no guarantee that all inputs will take this path.

We extend this method to the concurrency model of Creol and define the rules to actually compute the symbolic values for a given run. The formulas given in this section very closely resemble the rewrite rules of Creol's simulation environment [6], defined in rewriting logic [8] and implemented in Maude [2]. The rules are presented here in a slightly simplified manner to improve readability.

Denote by $\overline{s}$ the representation of program statements. Let $\sigma = \langle v_1 \triangleright e_1, v_2 \triangleright e_2, \dots v_n \triangleright e_n \rangle = \langle \overline{v} \triangleright \overline{e} \rangle$ be a map which records *key–value* entries $v \triangleright e$, where a variable $v$ is bound to a symbolic value $e$. The value assigned to key $v$ is accessed by $v\sigma$. For an expression $e$ and a map $\sigma$, define a parallel substitution operator $e\sigma$ which replaces all occurrences of every variable $v$ in $e$ with the expression $v\sigma$ (if $v$ is in the domain of $\sigma$). For simplicity, let $\overline{e}\sigma$ denote the application of the parallel substitution to every expression in the list $\overline{e}$. Furthermore, let the operator $\sigma_1 \uplus \sigma_2$ combine two maps $\sigma_1$ and $\sigma_2$ such that, when entries with the same key exist in both maps, the entry in $\sigma_2$ is taken. These operators are defined as equations in rewriting logic and are evaluated in between the rewrite steps.

In the symbolic state $\sigma$, all expanded variable names are bound to symbolic expressions. However, operations for method calls do not change the value of the symbolic state, but generate or receive *messages* that are used to communicate actual parameter values between the calling and receiving objects. Similar to the expressions bound to variables in the symbolic state $\sigma$, the symbolic representations of these actual parameters are bound in a map $\Theta$ to the actual and unique label value $l$ provided for each method call by Creol's operational semantics. Finally, the conditions of control statements along an execution path are collected in a list $\mathcal{C}$; the concatenation of a condition $c$ to $\mathcal{C}$ is denoted by $\mathcal{C}\hat{}c$.

The *configurations* of the rewrite system for dynamic symbolic execution are given by $\overline{s}\big[\Theta, \sigma, \mathcal{C}\big]$, where $\overline{s}$ is a run represented as a sequence of statements, $\Theta$ and $\sigma$ are the maps for messages and symbolic variable assignments as described above, and $\mathcal{C}$ is the list of conditions. The run $\overline{s}$ (as described in Section 2.1) is generated on the fly by the concrete rewrite system for Creol. Again, we concentrate on the statements for method calls and process release. A method `call` emits a message with the expressions for the method:

$$o_1 \overset{l}{\rightharpoonup} o_2.m(\overline{e}); \overline{s}\big[\Theta, \sigma, \mathcal{C}\big] \Longrightarrow \overline{s}\big[\Theta \uplus \langle l \rhd \overline{e}\sigma\rangle, \sigma, \mathcal{C}\big]$$

Because of the asynchronous behavior of Creol, the call might be received at a later point in the run (or not at all if the execution terminates before the method was selected for execution) by another rule, that handles the binding of a call to a new process and assigns the symbolic representation of the actual parameter values to the local variables in the new process ($\sigma \uplus \langle \overline{v} \rhd l\Theta\rangle$). The emission and reception of return values are handled similarly to call statements and call reception.

For conditionals, the local variables in the condition are replaced by their symbolic values ($\langle g\rangle; \overline{s}\big[\Theta, \sigma, \mathcal{C}\big] \Longrightarrow \overline{s}\big[\Theta, \sigma, \mathcal{C}\hat{}\langle g\sigma\rangle\big]$). This process is identical for the different kinds of conditional statements (**`if`**, **`while`**, **`await`**). The statement itself acts as a **`skip`** statement; it changes no variables and does not produce or consume messages. The resulting expression $g\sigma$ directly characterizes the equivalence class of input values that reach and fulfill the condition.

### 3.1 Application to Testing

Approaches to test case generation for structural coverage intend to find test sets that perform runs in the system for a specific coverage criterion. Two runs that cover the same parts of a system can be considered equivalent. A good test set should maximize the coverage, while minimizing the number of equivalent runs in order to avoid superfluous efforts in executing the tests.

Dynamic symbolic execution on a run gives the set of conditions that are combined to the path condition $\mathcal{C} = \bigwedge_{1 \leq i \leq n} c_i$ (for $n$ conditions), characterizing exactly the equivalence class of $t_{\mathcal{S}}$ that can repeat the same execution path. Only one test case that fulfills $\mathcal{C}$ is required. A new test case is then chosen to specifically avoid that a particular branch is taken by violating the respective $c_i$. To maximize decision coverage (DC), for instance, test cases have to be created such that for each of the conditions $c_i$, there is also a test case that violates this condition. The process of generating new test cases ends after all combinations required for the coverage criteria are explored.

More details and examples on how to use DSE to generate test cases in distributed systems can be found in the technical report to this paper [5].

## 4 Conclusions

The main contribution of this work is the novel extension of dynamic symbolic execution to non-trivial distributed and concurrent object models. This has been achieved by exploiting the properties of the Creol modeling language; in particular local scheduling control of the processes and strict encapsulation of the object state. This paper demonstrates how dynamic symbolic execution, combined with the executable architectural models of Creol, can be used to systematically derive equivalent input values, while avoiding the combinatorial explosion inherent in distributed concurrent systems. Our approach has been formalized in rewriting logic and implemented in Maude.

## References

1. R. S. Boyer, B. Elspas, and K. N. Levitt. Select-A formal system for testing and debugging programs by symbolic execution. *SIGPLAN Not.*, 10(6):234–245, 1975.
2. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, Aug. 2002.
3. F. S. de Boer, D. Clarke, and E. B. Johnsen. A complete guide to the future. In R. de Nicola, editor, *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 316–330. Springer, Mar. 2007.
4. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223. ACM, 2005.
5. A. Griesmayer, B. Aichernig, E. B. Johnsen, and R. Schlatte. Dynamic symbolic execution of distributed concurrent objects. Technical Report No. 408, UNU-IIST, March 2009.
6. E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and Systems Modeling*, 6(1):35–58, Mar. 2007.
7. J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
8. J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.
9. N. Tillmann and J. de Halleux. Pex - white box test generation for .NET. In *Proc. of the 2nd International Conference on Tests and Proofs (TAP'08)*, volume 4966 of *LNCS*, pages 134–153. Springer, 2008.
10. Y. Xie, A. Chou, and D. Engler. Archer: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM, 2003.